Appendix 2

ARTIFICIAL NEURAL NETWORKS: A HISTORICAL & TECHNICAL OVERVIEW

A2.1 Historical Background

Turing's Theory of Finite State Automata

In the 1950 the mathematician Alan Turing first proposed that the human brain functions as a computer. His theory was that both brains and computers belong to a class of machines, which he named 'Finite State Automata'. Furthermore he also posited that one of the properties of all Finite State Automata is that they are all capable of computing all computable functions. That is they all have the same computational potential. As such, anything that can be computed by one Finite State Automata can be computed by all Finite State Automata. An important corollary of all this is that it would therefore be possible to construct machines (Finite State Automata), which function exactly as the human brain, and therefore can emulate the brain and it's functioning, in part or full.

The importance of Turing's theory of Finite State Automata for the current thesis is that it led directly to efforts by researchers to attempt to emulate neural behaviour using machines. Collectively these efforts fall into an area of research that is now termed "Artificial Intelligence" or AI. Research on Artificial Neural Networks is a one of the approaches that has been used in the field of AI. The more general importance of

Turing's theory of Finite State Automata is that it predated, predicted and progenitated the development of electronic digital computing machines, more commonly known now as computers.

Hebb's Theory of Learning

When learning occurs in an animal, some change(s) must also occur somewhere within the nervous system of that animal to encode that learning. In the early 20th century, Sperry and others demonstrated that the human (and animal) nervous systems functionally consisted of cells which they termed "Neurons" and that these neurons were connected with one another [Dayhoff, 1990]. The connection between one neuron and another neuron was termed a "Synapse". Furthermore, it was discovered that neurons could transmit signals to one another via the synapses, though the direction of transmission of information at the synapse was uni-directional.

Attempting to explain the ubiquitousness of classical conditioning in animals, which contain a neural system, Hebb [1949] speculated that synapses were the most likely locations in a neural system where modifications could take place, because the synapses are the junctions between individual neurons where information is transmitted from one neuron to another. Specifically he suggested that learning is encoded by the weakening or strengthening of synaptic connections between neurons, rather than by any change(s) taking place within neurons or elsewhere in the nervous system. Changes at the level of the synapse, would effectively regulate how information was transmitted and therefore processed.

Hebb's Theory has been very influential in Psychology, but more importantly from the point of view of the current thesis, it served to motivate some of the directions taken by researchers who later investigated the development of artificial neural systems. These researchers sought to test Hebb's theory by constructing artificial neural systems incorporating modifiable synapses.

Rosenblat's Perceptron

Rosenblatt [1957] developed the *Perceptron*, a mathematically defined computer architecture, which emulates the function of a single neuron. Biological neurons function in the following fashion:

- A neuron receive inputs from other neurons by way of synapses

- These inputs can be excitatory (positive) or inhibitory (negative) and they vary in magnitude.

- The effect of these inputs is additive over a short period of time

- If the sum of all inputs (excitatory & inhibitory) exceeds a threshold then the neuron depolarises and fires. If the threshold is not exceeded depolarisation does not occur and the neuron does not fire. This known as the "all or none principle".

The Perceptron emulates a biological neuron. It does this mathematically by:

- Multiplying each input value by a specific weight (this is analogous to the connection strength of the synapse) to produce a weighted input value.

- Summing these weighed input values to produce a single value known as the activation value

- Performing a step function on the activation value to produce an output value. The step function produces a value of zero if the activation value is less than or equal to 0, or a value of 1 if the activation value is greater than zero.

The equation for the Perceptron is:

$$y = f\left(\sum_{i=0}^{n} a_i * w_i\right) \qquad \text{(A2.1)}$$

Where y is the output value

$f(x)$ = 0 if x <= 0, 1 if x > 0 (i.e. a step function)

i is an index value for inputs from 0 to n

a_i is the value input I

w_i is a weighting value used to weight input I

Note: there are n inputs, but n+1 (0,1…n) input values. When i = 0, the value of the input a_i , is always equal to 1, this is known as the bias input. Its purpose is discussed later.

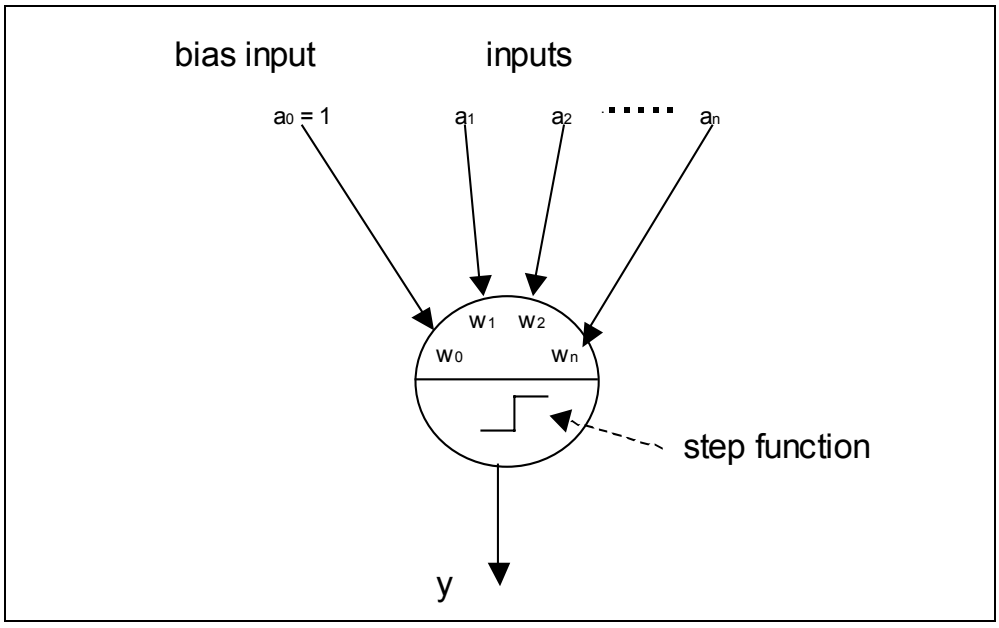


Figure A2.1 A Perceptron

Figure A2.1 above is a schematic representation of a Perceptron, which graphically portrays the computations presented in equation 2.1.

Rosenblatt and other early artificial neural network researchers, influenced by Hebb's theory, attempted to devise systems based upon the Perceptron, where the weights ($w_{0,}$ $w_{1, \ldots..}w_n$) could be adapted to produce a desired output in response to a particular input or set of inputs.

A2.2 Pattern Classification by the Perceptron

The basic function of a biological neuron is pattern classification. That is it divides all the possible patterns of inputs it receives, at its synapses, into one of two classes: the class of patterns that cause it to fire, and the class of patterns, which do not cause it fire. All patterns of inputs must belong to one or the other of these classes and cannot belong to both classes simultaneously, the "all or none" principle.

A *pattern,* which consists of the n input values to a Perceptron, can be mathematically represented as a point in n-dimensional space. The co-ordinates of a point, which locates the point in an n-dimensional *input space,* can be used to represent the measured attributes or features of an object. For instance, in several of the studies which are described later in this thesis, an individual's scores on various psychiatric symptom scales, are used to define a point in a multivariate data space (n dimensional input space) which represents that individual (the object). Each individual has a pattern of the scores (or other measures), and that pattern locates each individual at a specific point in the n-dimensional data space.

If the attributes, which are used to define each pattern, are related to the membership of each individual to one of the two classes (e.g. has a particular psychiatric diagnosis or does not have that diagnosis), then it can be expected that the patterns of individuals belonging to the same class will tend to be co-located in same general region of the n-dimensional space. If the two classes are mutually exclusive in the space, then they will each occupy a different and separate region of the n-dimensional input space. When this

situation occurs, and a straight line can be used to demarcate a boundary between them, then the two classes are said to be *Linearly Separable* in the n-dimensional space. Ripley [1994] defines linear separability: *"we say two groups are 'linearly separable' if there is a linear function of the variables which is positive for one group and negative for the other"* (p 116).

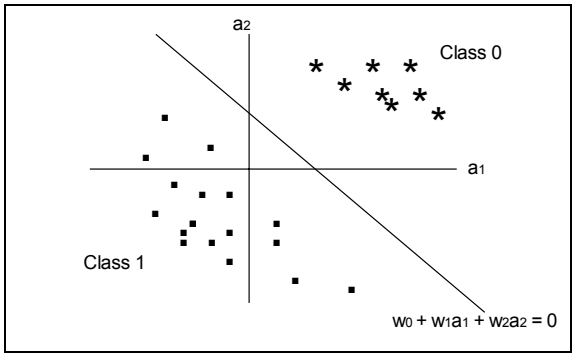


Figure A2.2 *A Linearly Separable Pattern Classification Problem in 2-dimensional space*

The 2-dimensional linearly separable pattern classification problem represented in Figure A2.2 shows that all the patterns belonging to Class 0 (★) are confined to one region of the space and all the patterns belonging to class 1 (■) are confined to another separate region of the space. A line ($w_0 + w_1a_1 + w_2a_2 = 0$) can be used to divide the space, so that all the patterns belonging to Class 0 fall on one side of the line and all the patterns belonging to Class 1 fall on the other. Note the need to include a term w0, which is effectively a constant needed to ensure that the line does not have to pass through the origin.

The Perceptron can be used to solve all linearly separable pattern classification problems, such as the one in Figure A2.2 above. A Perceptron with n inputs and an appropriate set of input weights, will always fire (output a value of 1) for all members of one class and fail to fire (output a value of 0) for all members of the other class. The appropriate set of weights (w0,w1,…wn) is the same set of values as the co-efficients (w0,w1,…wn) which define the separating line (or linear hyperplane in dimensions where n > 2).

Considering Figure A2.2, if each pattern's values on a1 and a2 (its co-ordinates in the above space) are used as inputs to a Perceptron, then a set of values can be found for the Perceptron's weights (in this case w0, w1 and w2), which cause the Perceptron to respond to each individual pattern with an appropriate response (0 for Class 0 and 1 for Class 1). The question then is, *how can an appropriate set of weight (w_0, w_1, …., w_n) be found which implements the desired solution to the classification problem?*

Rosenblatt's [1957] solution was *the Perceptron Learning Rule.* According to this rule, the Perceptron is first initialised with a random set of weights (that is the weight vector w_0, w_1, …., w_n .is populated with random values) Then each pattern (the input vector 1,a_1, a_2, ….., a_n) is presented to the Perceptron as inputs, one pattern at a time. For each pattern, the Perceptron uses equation 2.1 to calculate an output response, which is either a 0 or 1 (this is the equivalent of a biological neuron not firing or firing). For each such presentation of a pattern the resultant output represents a putative classification of that pattern into either Class 0 or Class 1 by the Perceptron. This putative classification is either correct or incorrect (note the assignment of one class as Class 0 and the other as

Class 1 is arbitrary). On each such *learning trial* (presentation of a pattern) the following formula is used to adjust the value of each weight (including the bias input weight w_0) :

$$w_i^{new} = w_i^{old} + \eta(y - t)a_i \qquad \text{(A2.2)}$$

Where η is the learning rate, a positive or negative fraction below 1

y is the output generated by the pattern, its putative class

t is the true or target output value, which denotes the pattern's class

i is an index for inputs

a_i is the value of input I

According to equation A2.2, the updated weight (w^{new}_i) is the previous weight (w^{old}_i) plus the product of η (the learning rate), the difference between the correct output and the putative output ($y - t$), and the input value (a_i). Thus when the putative output is the same as the correct output then their difference ($y - t$) is zero and the updated weights for that pattern remain the same as the old weights (i.e. there is no change). On the other hand when the putative output and the correct output are not the same, then their difference is either 1 or –1. In this case the weights for that pattern are each changed by the addition or subtraction of an amount in proportion to the size of the input value for that weight.

Each case in the training set of patterns (cases) is usually processed many times. Cases in the training set are processed one by one from first to last. When the end of the training set is reached, processing recommences at the start. Each pass through the training set is

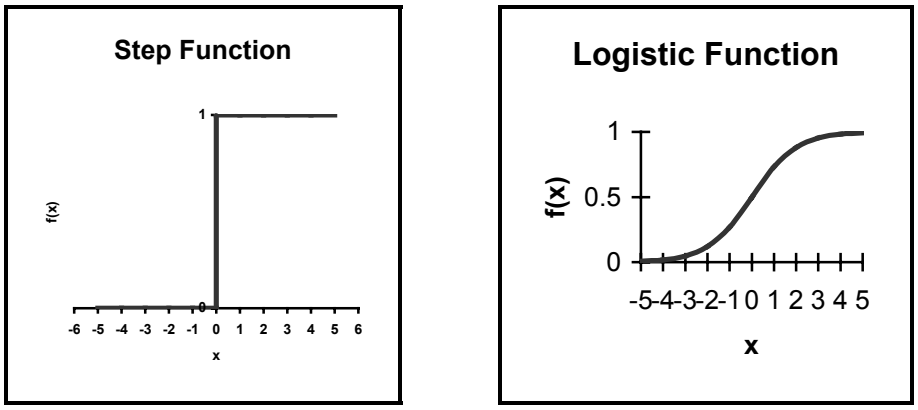
called a learning *trial*. Processing continues for a number of trials, until the Perceptron classifies all the patterns correctly.

Rosenblatt [1957] was able to show, both theoretically and empirically, that using this learning rule a Perceptron would always develop an appropriate set of weights in a finite number of trials, provided that the two classes of input patterns are linearly separable in the n-dimensional input space. This holds for any value of n. Rosenblatt's findings were important because they showed that under certain conditions Perceptrons could be used to discover a linear Discriminant function from data (a set of patterns with known class membership).

However, Minsky and Papert [1969] in an influential review of research on Perceptrons up to that date, pointed out that both theoretically and empirically, it could be demonstrated that Perceptrons could not solve problems where the two classes of input patterns are not linearly separable. Since many real-world classification problems are not linearly separable and biological neural networks can also be shown to be able to learn classifications, which are not linearly separable, they concluded that further research on Perceptrons was a dead end, not likely to yield any practically useful results. However as we will see in the next few sections, there are extensions, which can be made to the basic Perceptron, which not only overcomes this limitation, but also yields some mechanisms for solving complex classification problems.

A2.3 The Logistic Activation Function

The *activation function* used in Rosenblatt's [1957] Perceptron is a step function. The step function is a scalar to scalar function, which abruptly changes output value from 0 to 1 at a threshold of 0 on the input value. The logistic function (also called a sigmoid function) is also a scalar to scalar function which performs a similar mapping of input to output, but changes output values smoothly from 0 to 1, between an interval of about –5 and + 5 on the input scale. See Figure x below



$f(x) = (if\ x <= 0,0),\ (if\ x > 0,1)$ (A2.3)

$$f(x) = \frac{1}{1+e^{-x}} \qquad \text{(A2.4)}$$

Figure A2.3 *The step function and the logistic function*

By substituting the logistic function for the step function as the activation function of a Perceptron, then the Perceptron is identical to a Linear Logistic Discriminant and similar to a Logistic Regression [Sarle, 1994, Bishop, 1995, Reed & Marks 1999]. The Linear Logistic Discriminant has been widely studied and used in the statistics literature [Bishop, 1995]. It is generally more useful in applications to linear classification problems than Rosenblatt's Perceptron for several reasons.

Firstly the use of a logistic activation function allows for the use of optimization algorithms which can be used to find an optimum set of weights, which minimizes misclassifications in problems where the distributions of the classes in the input space are partially overlapping. Recall that Rosenblatt's Perceptron Learning Rule can only be applied to problems, where the class distributions are non-overlapping.

Secondly, unlike the step function, the output of a logistic function (and therefore the output of a Perceptron with a logistic activation function) can vary between 0 and 1 rather than only take on values of only 0 or 1. Under certain conditions (to be discussed later), the case-wise output of an optimized Logistic Perceptron can be used as the Bayesian Posterior Probability that the case belongs to a particular class. Knowing the probability of class membership is more useful than knowing a putative class assignment. Posterior probabilities can be combined with information about the prior probability of class membership in a particular application setting and so as to maximize classification accuracy in that setting [Ripley 1994, Bishop 1995]. Furthermore extending this framework, a Loss Matrix can also be applied, along with the prior and posterior probabilities to make classification decisions, which take into account a number of criterion and factors.

A2.4 The Perceptron Error Surface

One way to understand how a Perceptron can implement an optimal linear decision boundary between two classes is to consider the relationship between variation in the weights and variation in the Perceptron's error.

The Perceptron Learning Rule discussed in Section 1.3.1. calculates the case-wise error as the difference between the Perceptron output for that case (its putative classification, *z* in equation A2.2) and the true classification for that case (also known as the target value, *t* in equation A2.2). On the other hand, a good measure of error across a whole set of cases (such as the training set) is the Mean Square Error (or MSE), which can be calculated as:

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - t_n)^2 \qquad \text{(A2.5)}$$

Where N = the number cases in the set

n = an index for cases in the set ranging from 1 to N

y_n = the Perceptron's putative output for case n

t_n = is the true or target output for case n

If we, again, consider the linearly separable classification problem presented in Figure 2, and assume a Perceptron with a logistic activation function, then a plot of MSE as a function of variation in the two main weights (w1 and w2), will have a general shape like that presented in Figure A2.4. below.

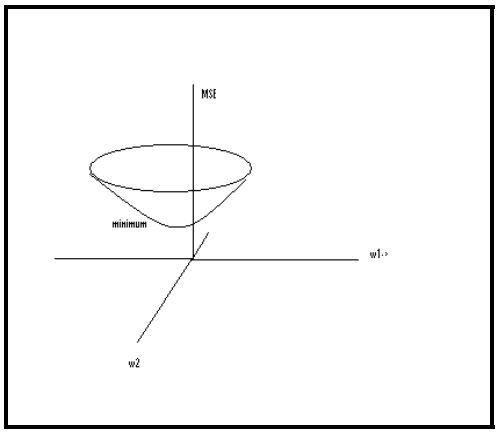


Figure A2.4 *Perceptron Error Surface.*

As can be seen from Figure 2.4 above, for a Perceptron there is a single unique combination of weights at which MSE is at a minimum. Furthermore, the general overall shape of the relationship between weights and MSE is that of a basin (Bishop 1995). The general shape of this relationship can be generalized to any number of weights (and therefore inputs). In weight spaces with a dimensionality of greater than 2, the shape of the relationship is a hyper-basin.

The basin shape of the Weight-Error relationship, with its property of a single minimum point, makes it easy for optimization algorithms to locate this minimum for a given set of

training cases (and therefore to identify the set of values of the weights, which maximize the accuracy of classification of cases contained in the training set).

A2.5 The Multi-Layer Perceptron

The important limitation of the Perceptron is that it can only learn to classify patterns, which have a linear classification boundary as a best solution. That is a straight line (or a plane in higher dimensions) can form a boundary between the two classes, which maximally separates the two classes, so that on one side of line the majority of patterns belonging to one class and on the opposite side of the line the majority of patterns belong to the other class. If the patterns are linearly separable, then all the patterns on one side of the line belong to one class and all those falling on the opposite side belong to other class.

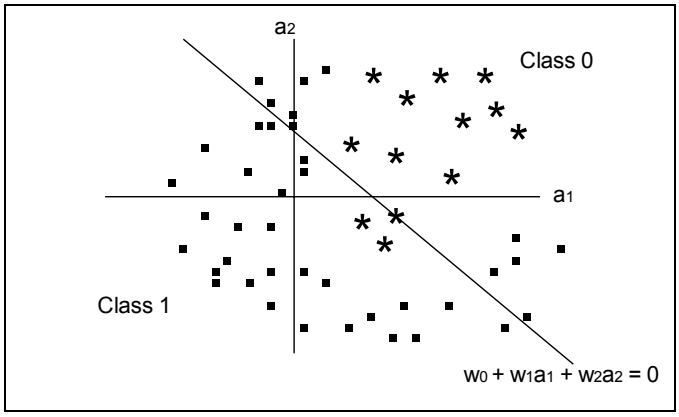


Figure A2.5 *Non Linearly Separable Classes*

Figure A2.5 above gives an illustration of a classification problem where the two classes are not linearly separable. That is it not possible to find a straight line, which can be used to separate the two classes. Therefore, the Perceptron is not able to solve this type of

classification problem. However the two classes in Figure 2.5 can be separated by a non-linear boundary. That is a curve or a combination of several straight lines. This type of classification problem can be solved by a Multi-Layer Perceptron [Rummehart & McCelland, 1983], which consists of several Perceptrons arranged in a network with at least 2 layers, and with interconnections between the layers.

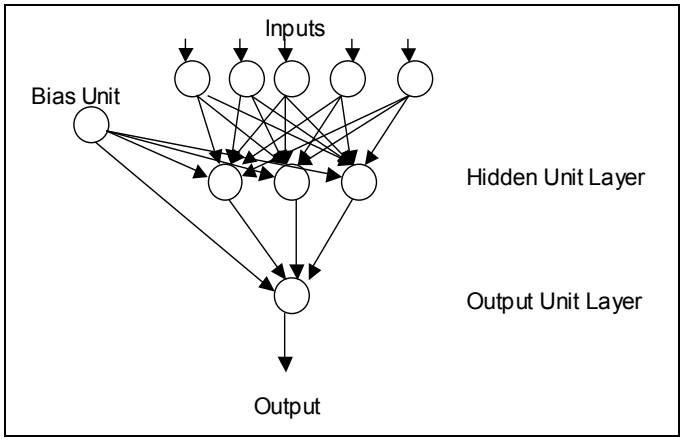


Figure A2.6 *A Multi-Layer Perceptron*

Figure A2.6 above presents a Multi-Layer Perceptron. Each of the circles represents a unit, which is a single Perceptron-type processing unit. Arrows represent the flow of information.

Inputs

Units in the input layer simply pass on the input value unchanged. There is no transformation of the input value through a step function or any other function. Each unit in the input layer has a connection to all units in the hidden layer.

Hidden Layer

Units in the hidden layer (called so because it is hidden from direct connection to the outside between input and output units) are Logistic Perceptrons. Each hidden layer unit receives an input from each unit in the input layer. Additionally each hidden layer unit receives an input with a value of 1 from a bias unit. Each unit in the hidden layer performs a weighted sum on its inputs. For unit j the formula for this summation is given as:

$$s_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \qquad \text{(A2.6)}$$

Where j is an index for hidden units

i is an index for inputs

d is the number of inputs

s_j is the activation value for unit j

x_i is the value of input I

$w^{(1)}_{ij}$ is the weight of the connection of input i to unit j in layer 1

This is sometimes referred to as the unit's *Combination Function.* The result of this summation, commonly referred to as the unit's *Activation Level* is then passed on to an *Activation Function* (which is usually a logistic function) to arrive at an *Output Value* for the hidden unit:

$$z_j = f(\sum_{i=0}^{d} w_{ji}^{(1)} x_i) \qquad \text{(A2.7)}$$

Where z_j is the value of the output from hidden unit j

f() is the unit's activation function, usually a logistic function

Output Units

Units in the output layer (there can be one or more, but in this thesis we are mainly only concerned with the case where there is only one), receive as their inputs, the outputs of the hidden layer units. Like the hidden layer units, an output layer unit performs a weighted sum on the inputs and then passes on this sum to a logistic activation function for calculation of an output value. Thus the Output Unit is also a Logistic Perceptron. The output of the output unit is given by the formula below, which is similar to (2.5) above.

$$y_k = f(\sum_{j=0}^{m} w_{kj}^{(2)} z_j) \qquad \text{(2.8)}$$

Where Y_k is the value of the output from Output Unit k
$w^{(2)}_{ij}$ is the weight of the connection of input i to unit j in layer 2
m is the number of hidden units

A2.6 Equation for a Multi-Layer Perceptron

Substituting (A2.7) for z_j in (A2.8), the full formula for an MLP with one hidden layer and one output unit is:

$$y = f(\sum_{j=0}^{m} w_{kj}^{(2)} f(\sum_{i=0}^{d} w_{ji}^{(1)} x_i)) \qquad \text{(A2.9)}$$

Where

i	is an index for inputs
d	is the number of inputs
x_i	is the value of input I, when I=0 then $x_i = 1$
y	is the putative output value generated by the network
j	is an index for hidden units
m	is the number of hidden units
$w^{(1)}_{ij}$	is the weight of the connection of input i to hidden unit j in layer 1
$w^{(2)}_{kj}$	is the weight of the connection of hidden unit j to output unit k in layer 2
$f()$	is the activation function, $f(x) = 1/(1+e^{-x})$

(A2.9) directly defines the Multi-layer Perceptron. It relates the output (y) for a case to an input pattern vector (x0, x1, …. Xd) for that case. The total number weights (parameters) in this neural network is (d+1)*m+m+1.

For example, for the MLP depicted in Figure 4. in which d = 5, m=3 and there are 22 ((5+1)*3+3+1) adaptable weights.

A2.7 Classification with an MLP

The MLPs advantage over the simple Perceptron is that it can be used to generate solutions to non-linearly separable classes, such as the one depicted in Figure A2.7 below:

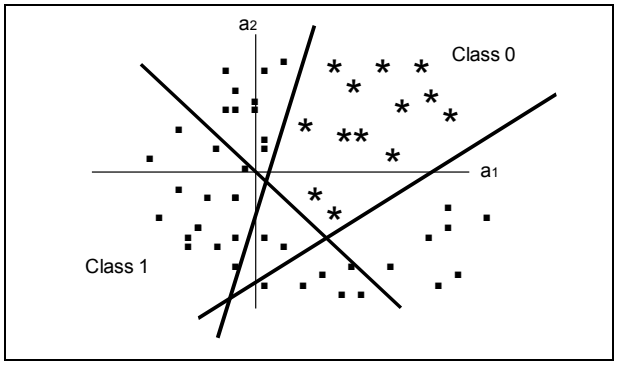


Figure A2.7 *MLP solution to a problem involving non-linearly separable classes*

The simple Perceptron can form only linear half-plane decision regions. That is it can only halve a space into two regions using a straight line or linear hyperplane. If the classification problem being addressed is linearly separable, then this capability is enough to solve the problem. MLPs on the other hand can form more complex, bounded and unbounded convex polygon, decision regions, usually referred to as convex hulls (Lippman, 1987). These convex decision regions are formed by the intersection of linear half-plane decision regions formed by each unit in the MLPs hidden layer.

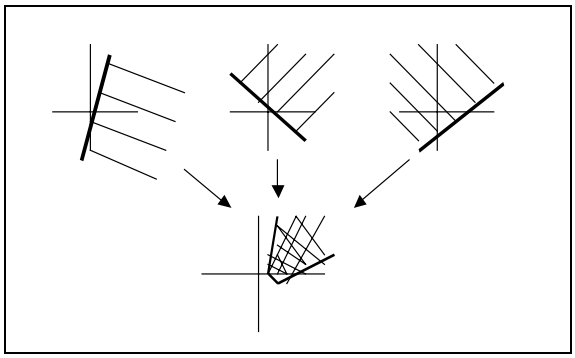
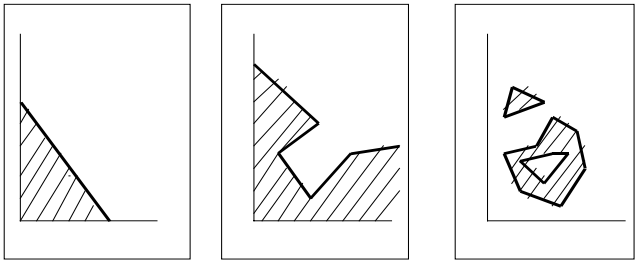


Figure A2.8. *The half-plane decisions regions of 3 hidden units are combined by the output unit, to form the convex hull decision region of a Multi-layer Perceptron, required to solve the classification problem in Figure 2.7 (adapted from Reed & Marks 1999)*

In Figures A2.7 and A2.8 above, the three lines collectively form a convex hull, which demarcates objects belonging to class 0 from those belonging to class 1. The MLP required to solve this problem would have 2 input units, 3 hidden units and 1 output unit. For each hidden unit, its weights define the equation of one of the lines. The function of the output unit is to combine the outputs of the 3 hidden units and therefore assign each pattern to either class 0 or class 1.



(a) Perceptron **(b) MLP few hidden units** **(c) MLP many hidden units**

Figure A2.9 *Decision boundaries (a) a Perceptron, (b) an MLP with a small number of hidden units (c) an MLP - large number of hidden units. Adapted from (Bishop 1995).*

Figure A2.9 demonstrates that by adding hidden units, successively more complex MLPs, can be constructed, which have successively more complex class decision boundaries.

A2.8 Multi Layer Perceptron Error Surface

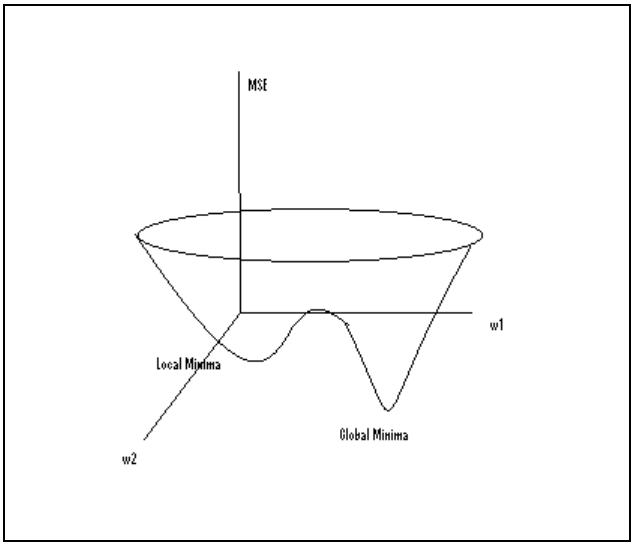


Figure A2.10 *Multi-layer Perceptron Error Surface. Adapted from Bishop (1995)*

Figure A2.10, above, depicts the Salient features of the error surface of a Multi-Layer Perceptron which is obtained by plotting the networks MSE as function of variation in the values of the weights. For the purposes of exposition, only two weight axes are shown. In reality an MLP would have a weight space with a much higher dimensionality. The true error surface of an MLP would be a function of all the weights, not just two. The error surface of a single Perceptron is generally a basin, with a single minima. The error surface of an MLP is generally a more complex manifold with possibly many minima separated by saddle points. One of these many minima is the global minima. That is the point on the error surface with the lowest possible MSE. All the other minima are local

minima, which by definition have an MSE which is greater than that of the global minima.

The values of the weights associated with Global Minima represent the best possible set of weights for the MLP to maximally classify cases to their correct class. The weights associated with Local Minima may also represent relatively good solutions, which may give classification accuracies not far worse than those obtained at the Global Minima. Minima on the error surface are generally separated by saddle points (also called local maxima). The MSE of saddle points is generally relatively high. The weights associated with saddle points and other regions of the error surface with a high MSE represent sets of weights, which are poor solutions to classification problems by the MLP.

Thus in order to find the best solution to a classification problem using an MLP, we need a method for finding the set of weights associated with the Global Minima of the MLPs error surface. If we are only interested in relatively good solutions, then we only need a method, which can locate local minima. As we shall see there are, available, several methods (algorithms) which, can be used to locate minima.

A2.9 Back-Error Propagation

The algorithm that is most often used to locate a minima on the Error Surface of a Multi-Layer Perceptron, and therefore to locate a set a weights which can provide relatively good solution to classification problem is called Back Propagation of Errors.

It was known in the 1960s that MLP type arrangements of Perceptrons into interconnected layers could be used to solve non-linearly separable classification problems (Dayhoff, 1990). However what was not known, at that time, was a method for finding an appropriate set of weights for the MLP as a whole, given a training set of cases. The Perceptron Learning Rule cannot be applied to an MLP because it relies upon correcting the weights by an amount proportional to the error. The hidden units of an MLP (which by definition are not connected to outputs) have no correct response against which they can compare their activations and ascertain a level of error. Thus the Perceptron Learning Rule, which can be used adjust the weights of output units, cannot be used to adjust the weights of the hidden units.

What is needed is a way to ascertain error for hidden units. As well, the error needs to be individually ascertained for each hidden unit, because whilst one hidden unit may need to make large weight adjustments because it is way off target, a neighbouring hidden unit may only need to make relatively small weight adjustments, because it is very close to it's target. This became known as the C*redit Assignment Problem* (Bishop, 1995).

A solution to the Credit Assignment Problem, Back-Propagation of Errors (BackProp), seems to have be independently discovered by several investigators, but is generally attributed to Werbos (1974). One of basic insights for BackProp is that for an MLP, as with the Perceptron, for every possible set of values of the weights there exists a corresponding error value, which measures how well that particular set of weights segregates the cases in the training set into two classes. The lower the error, the better the segregation. The goal of the BackProp algorithm is to find a particular set of weights, which produces a relatively minimum value of the error and therefore a relatively maximum segregation of cases into classes.

The Error Surface of an MLP (The surface which results from plotting error for a particular dataset as function of all possible values for the weights: Figure 9) is generally more complex than the corresponding Error Surface of a Perceptron (Figure 4). The MLP Error Surface can contain many minima, one of which will be a global minima (the best possible solution) and the others being local minima, some of which represent relatively good solutions (almost as good a the global minima), whilst others would be relatively poor solutions (much worst than the global minima).

BackProp is a gradient descent algorithm, which will usually find a good minima. (Bishop 1995, Reed & Marks 1999) It starts at random point on the error surface (weights are initialised to a set of random values). It then works out which direction from this point is a downhill direction on the error gradient (that is a direction which points towards a minima) and then takes a small step in this direction, by making small

individual changes to each weight. From this new point on the Error Surface, BackProp again works out which direction is pointing down the error gradient (towards a minima), and again takes a small step in this new direction. This procedure of repeatedly taking small steps downhill from wherever point it is at, continues until BackProp reaches a point where no step in any direction is a downhill step (which by definition is a minima).

The general concept of searching for a minima by gradient descent on an Error Surface is for the most part a simple idea. The difficult and critical part, which Werbos (1974) was able to solve, is how to determine the direction of each step, so that the step is in a downhill direction on the local error gradient and therefore in the general direction of a minima. Since the Error Surface is a function of the weights, then the direction that needs to be determined is a direction in weight space (that is a space with dimensionality M, where M is the total number of weights in the MLP and the axis of this space are the individual weights).

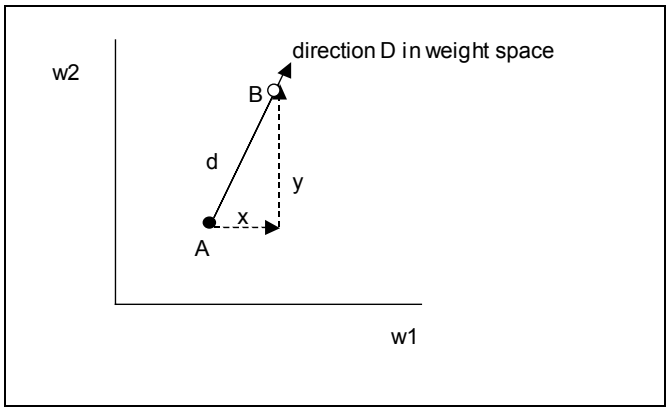


Figure A2.11 *Direction and movement in a two dimensional weight space*

The concept of direction and movement in weight space is demonstrated in Figure A2.11 above, using a simplified two dimensional weight space. For an MLP, the actual dimensionality of the weight space (M) will be much higher, but all the principles illustrated in Figure A2.11 still apply. From point A (black dot), movement of any distance in any direction in 360 degrees is possible. A movement from A to B (white dot) is a movement of distance d in direction D. This movement can be decomposed into two constituents: a movement of distance x in direction w1 and a movement of distance y in direction w2. In the higher M dimensional weight space of an MLP a movement from a point A to a point B can similarly be decomposed into M movements of set distances in directions parallel to the M weight axis.

The BackProp algorithm takes a step (makes a move) in weight space by individually determining the magnitude of the distances that it needs to move in the direction of each weight axis. The goal of the BackProp algorithm is to always take a step in the general direction of an Error minima. To accomplish this the algorithm needs to make larger distance moves along weight axis that have a steep error gradient (i.e. give more reduction in error for a given change in weight) and smaller distance moves along weight axis, which have less steep error gradients. The size of the move along any particular weight axis is proportional to the steepness of the error gradient Therefore the BackProp algorithm needs a method(s) for calculating (or estimating) the error gradient for each of the M individual weight. That is a method for determining the amount of change in error

that will result from a given change in the value of a weight. Mathematically this ratio is called the derivative of the error with respect to the weight:

$$\frac{\Delta E}{\Delta w} = \frac{dE}{dw} \qquad \text{(A2.10)}$$

Where ΔE is the change in error

Δw is the change in weight

$\frac{dE}{dw}$ is the derivative of error with respect to the weight w

Because the direction given by this method is only an estimate of the general direction of a minima, then the BackProp algorithm, conservatively, only takes a small step in this direction. That is it only makes small change to each weight, which is proportional to the estimated magnitude error gradient of that weight at that point on the Error Surface. The basic weight update rule for BackProp algorithm is:

$$\Delta w = -\eta \frac{dE}{dw} \qquad \text{(A2.11)}$$

Where η is a learning rate parameter, which is usually a small fraction e.g. 0.1

Δw is the amount by which the weight is changed

Werbos (1974) and others have shown that, provided the individual unit's activation functions are differentiable (the logistic function is a commonly used differentiable activation function), then the *chain rule for derivatives* , which is a basic technique in algebra, can be applied to efficiently calculate the value of the error derivative for each weight. This inturn allows the BackProp algorithm to calculate a value by which to change each weight (using equation A2.11 above), so that the resultant small step it takes on the error surface is in the general direction of a minima.

Using the chain rule, the derivative of error with respect to a given weight can be decomposed into two, simpler (to calculate), derivatives: the derivative of error with respect to the units activation level and; the derivative of this activation with respect to that given weight. Applying the chain rule:

$$\frac{dE}{dw_{ji}} = \frac{dE}{da_j} . \frac{da_j}{dw_{ji}} = \delta_j z_i \qquad \text{(A2.12)}$$

Where $\frac{dE}{da_j} = \delta_j$ and $\frac{da_j}{dw_{ji}} = z_i$

i is an index for inputs

j is an index for units

E is a measure of error for the whole network

a_j is the activation level for unit j, $a_j = \sum_i w_{ji} z_i$

w_{ji} is a weight for the input from i to unit j

z_i is the value of the input from i

δ_j is the error of a unit j, it is calculated differently for output units versus hidden units

We already have values for z_is so all we need to do is calculate the values for δ_js. For an output unit, $\delta_{k\ is}$ calculated by again using the chain rule to substitute partial derivative:

$$\delta_k = \frac{dE}{da_k} = \frac{dE}{dy_k} \cdot \frac{dy_k}{da_k} = (t - y_k) f'(\sum_j z_j w_{kj}) \qquad \text{(A2.13)}$$

Where $\frac{dE}{dy_k} = (t - y_k)$ and $\frac{dy_k}{da_k} = f'(a_k) = f'(\sum_j z_j w_{kj})$

$f'( )$ is the derivate of the activation function

$f'(x) = f(x)(1 - f(x))$ for a Logistic Activation Function

t is the true or target value of the output from the training set

y_k is the output value for output unit k

j is an index for hidden units

z_j is the output of hidden unit j

w_{kj} is the weight connecting hidden unit j to the output unit

k is an index for output units, usually there is only 1 and k=1

For hidden units, δ_j can be calculated, by again applying the chain rule to substitute simpler (to calculate) partial derivatives:

$$\delta_j = \frac{dE}{da_j} = \frac{dE}{da_k} \cdot \frac{da_k}{da_j} = w_{kj} \delta_k f'(a_j) \qquad \text{(A2.14)}$$

Where $\frac{dE}{da_k} = w_{kj} \delta_k$ and $\frac{da_k}{da_j} = f'(a_j)$

δ_k is the error of output unit k to which hidden unit j sends its output

w_{kj} is the weight of the connection from hidden unit j to output unit k

a_j is the activation value for the output unit

In summary then, the algorithm for Back-Error Propagation is:

- Start with a set of weights initialised to random numbers
- Examine the first case in the training set
- For the current case forward propagate from inputs through to the output unit to arrive at a putative output, using equation (A2.9)
- For the output unit compute an error signal $\delta_{k,}$ .using equation (A2.13)
- For each hidden units compute an error signal $\delta_{j,}$ using equation (A2.14)
- Adjust all the weights in the output unit and all the hidden units using the formula:

$$w_{ij}^{new} = w_{ij}^{old} - \eta \, \delta_j z_i \qquad \text{(A2.15)}$$

Where H is a learning rate parameter, which is usually a small fraction e.g. 0.1

δ_j is the error of unit j

z_i is the output of unit I (which is feeding into unit j)

w_{ij} is the weight to unit I from j

j is an index for hidden units

- Proceed with the next case in the training set, repeating 3,4, 5 and 6 above.
- Continue processing all the cases, starting again at the first case after the last case has been processed (one pass through the training set is called a learning trial), until the MSE reaches a predetermined minimum or some other criteria is reached (eg. Maximum number of learning trials)

Unlike the Perceptron learning rule, Back-Error Propagation is not guaranteed to always converge to the best solution. This is because it is possible for the algorithm to find a local minima rather than the global minima. Whether or not the algorithm finds a local minima or the global minima depends upon three things: the nature of the Error Surface for a particular problem (the number and distribution of minima); the starting point (the randomly set initial values for the weights), and the learning rate. As we will discuss later, all three of these can be addressed so as to maximise the chances of finding a good solution.

The Back-Error Propagation algorithm is a Gradient Descent algorithm. Each time the algorithm changes the value of a weight, it does so by using information about the gradient of that weight with respect to error (the derivative of the error with respect to the weight is this gradient) to take a step (of size η) in the direction of reducing error. By changing all the weights in the MLP in the direction of reduced error, each back-propagation step is a step downhill (descending the error gradient) from whereever it is before taking that step. If the goal is to minimize error on the training set then an appropriate place to stop is when taking a step always results in going uphill (increased error). This stopping point could be the "Global Minimum" (the point of absolute minimum error), which is best possible set of weights, for solving the classification problem, presented by the training set. It could also be a "Local Minimum" (a local low point in the error surface, which is not the lowest possible). Some local minima may provide a relatively good solution, even if it is not the best possible solution, for that training set. If the starting point is, by chance, uphill from a local minimum with very

steep sides, then the algorithm will simply walk downhill into it and stop there. If the sides are not very steep, the algorithm may be able to step out of that minima and continue onwards to search for a better minima.

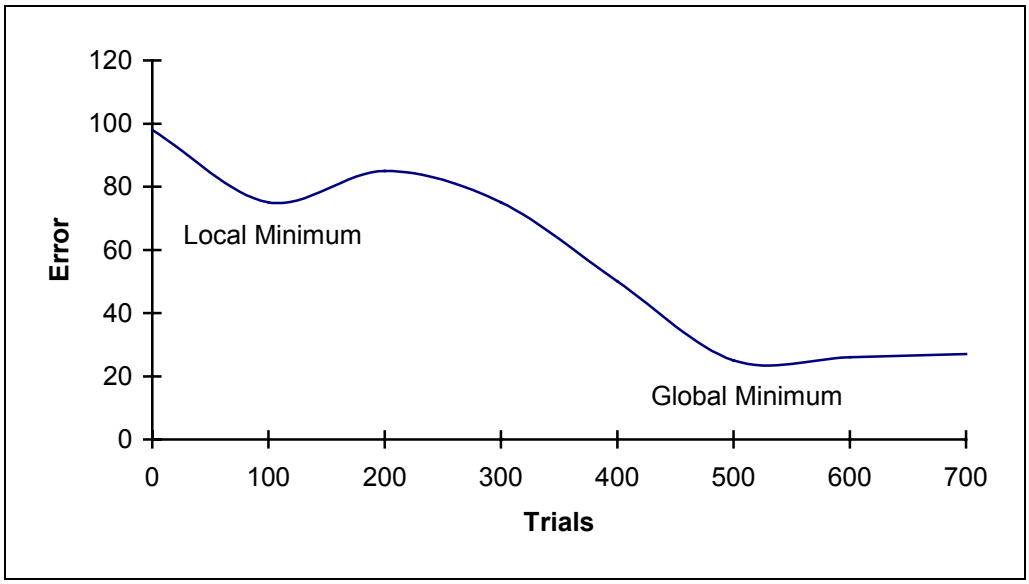


Figure A2.12 *Training Set Error by Trials of Learning. In this example, as training progresses the Back-Error Propagation algorithm encounters a local minimum at about 100 trials, which it passes through and eventually finds a global minimum at about 500 trials.*

The graph in Figure A2.12 can be directly associated with the error surface depicted in Figure A2.10. If the initial random small weight values start the algorithm high up the left wall of the local minima, then by using gradient descent, the Back-Prop algorithm may follow a course along the error surface depicted in Figure A2.10, which traces the path (in terms of change in error over trials), depicted in Figure A2.12. In this case the global minima, found at 500 trials in Figure A2.12 above, corresponds to the Global Minima depicted on the right hand side of the error surface of Figure A2.10.

Figure A2.13 below demonstrates the trajectory of the BackProp algorithm through a weight space (defined by w1 and w2), and across a corresponding Error Surface, from a starting point on the edge of the error surface, towards an error minima. The concentric ellipses are iso-error contours, centred upon the minima. The trajectory in Figure A2.13 can be associated with a segment of the learning curve in Figure A2.12 ranging from 300 to 500 trials.

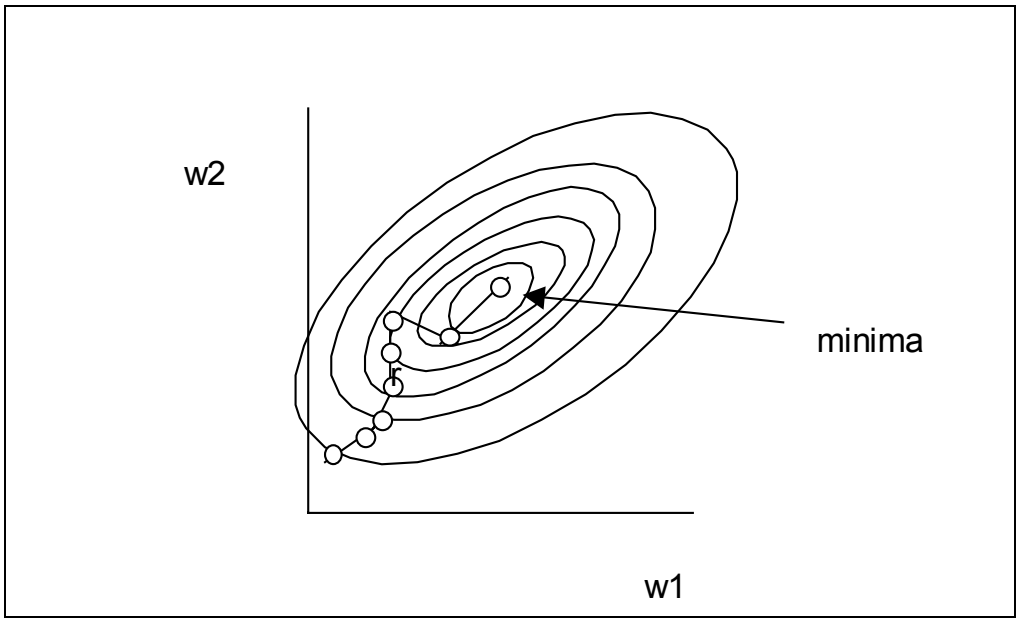


Figure A2.13 *Trajectory of the BackProp algorithm through weight space, towards an error minima. Adapted from Reed & Marks 1999.*

The learning rate (η) determines the size of each step the algorithm takes. If the step size is too small, then gradient descent makes it more likely that the algorithm will walk into a

depression in the error surface (i.e. a local minimum), which it can't step out of (because the step size is too small). Too large a learning rate can also be problematic. Taking large steps will help avoid local minima by stepping over them and through them, but it may also make the algorithm step out of the gradient leading to the global minima and thus make it difficult to converge to the global minima. This will tend to make the algorithm walk around the error surface always missing the global minima (and local minima) and never find a good solution.

As will be discussed later there are steps than can be taken to condition the problem, which improve the likelihood of finding good solutions.

A2.10 Modifications to Back Prop

Online Vs Batch Learning

The description of Back-Propagation given above is for the online version of the algorithm, which updates the network weights after processing each case. There is another version, the Batch Processing version, which updates the weights after processing all the cases in the training set in one pass. This is achieved by accumulating δs and zs from each case and substituting in equation (A2.15) to get:

$$w_{ij}^{new} = w_{ij}^{old} - \eta \sum_{n} \delta_j{}^n z_i^n \qquad \text{(A2.16)}$$

Where n is an index for cases in the training set

The fundamental difference between online and batch learning is that in online learning there are N steps (where N is the no. of cases in the training set) along the error surface for every one step taken by batch learning. The single batch step is a summated step of N-steps that would be taken by online processing. However the paths along the error surface taken by these two versions of the Back-Prop algorithm can be very different. This is because, the size and direction of each step is determined locally for each weight and takes place from a location that is determined by where the last step left off. Both versions will locate minima, but because they take different paths may locate different minima.

Momentum

Momentum involves the addition of a term to the weight update rule (A2.15 or A2.16), which is proportional to the size of the last weight update:

$$w^{new} = w^{old} + \Delta w^{new} + \alpha \Delta w^{old} \qquad \text{(A2.17)}$$

Where $\Delta w^{new} = -\eta \delta z$ as in (A2.15) for the current weight update

$\Delta w^{old} = -\eta \delta z$ as in (A2.15) for the previous weight update

α is the momentum term a fraction between 0 and 1, which determines, the proportion of the previous weight update that is added to the current weight update

Thus if the last weight update was by a large amount, the weight is updated by a greater amount. If the previous weight update was small then only a small amount is added to the weight update. This has the effect of smoothing the trajectory of Back-Prop over the error surface. It keeps the trajectory going in the same general direction as the previous step, and prevents abrupt changes in direction taken in just one or a few steps.

Momentum also has the effect of amplifying the learning rate, but in a proportional sense. When the algorithm has been taking large steps, then it will continue to do so for a few more steps, but as the steps get smaller, momentum also gets smaller. This helps the BackProp algorithm to escape from local minima, but to stay in the vicinity of the global minima, because local minima are shallower, thus improving the chances that training will terminate at the global minima.

QuickProp

QuickProp is an algorithm developed by Fahlman (1987). Like BackProp it is a locally adaptive method. That is it makes changes to individual weights in an attempt to move from the current position towards a minima on the error surface. Where it differs is in how it calculates the weight change amount. QuickProp makes two important assumptions:

- The error surface for each weight, from any point on the error surface, can be approximated by a parabola that opens upwards.

- The change in slope of the error for each weight is independent of the other weights that change at the same time.

The basic weight change rule for QuickProp is:

$$\Delta w(t) = \frac{\frac{dE}{dw}(t)}{\frac{dE}{dw}(t-1) - \frac{dE}{dw}(t)} . \Delta w(t-1) \qquad \text{(A2.18)}$$

Where $\Delta w(t)$ is the current weight change amount

$\Delta w(t-1)$ is the previous weight change amount

$\frac{dE}{dw}(t)$ Is the derivative of the current weight with respect to error and is calculated the same as in BackProp using equation (2.10)

$\frac{dE}{dw}(t-1)$ Is the derivative of the previous weight with respect to error and is calculated the same as in BackProp using equation (2.10)

This weight update rule rests upon two relations:

Firstly , $\frac{\frac{dE}{dw}(t-1)-\frac{dE}{dw}(t)}{\Delta w(t-1)}$ which is a finite difference approximator of the second derivative, and secondly the equation as whole approximates Newton's method for finding the minima of a one-dimensional function, which is:

$$f(x):\Delta x=-\frac{f'(x)}{f''(x)} \qquad \text{(A2.19)}$$

As a result, what the QuickProp algorithm is attempting to do, is to locate the nearest minima in one single step. If the assumption that error surfaces has a multivariate upward parabolic shape is correct it will find it. Mostly though, this is not exactly correct and QuickProp either overshoots or undershoots the minima. If it overshoots (sign of the first derivative changes) it then moves backwards to a point intermediate between the previous position and the current position. If it undershoots (sign of the first derivative is the same as previous) it will take another step forward, by recalculating A2.18 from this new point. Normally on an undershoot, the value of the first derivate should reduce (as it would if the local error surface is parabolic) and so the next step size will be smaller. If the value of the first derivative is the same as larger as in the previous step, it indicates that the assumption of a local parabolic error surface is grossly incorrect and QuickProp will then default to gradient descent for that move. As well there is a maximum growth factor

parameter (usually 1.75), which limits a weight change to that growth factor times the previous weight change.

A2.11 Other Optimisation Algorithms

The popularity of BackProp, QuickProp and other variants of Back Error Propagation for the selection of a set of the weights in an MLP that minimize error, ignores an extensive body of research work, which normally goes under the name of numerical optimisation (Sarle, 1999). These other optimisation algorithms are generally much faster, and often they locate the global minima of a training set more accurately. However they are all affected by ill conditioning and there is no evidence that they generalize any better than the Props (Bishop 1995, Sarle 1999, Reed & Marks 1999). As well the Props are the only algorithms which can be implemented locally, at each Perceptron type unit. All the others require network wide information. This has important implications for possible hardware implementations of neural networks. It allows for the development of scalable parallel distributed information processing hardware, which can be built up by interconnecting any number of individual hardware implementations of Perceptron type units.

A2.12 Universal Function Approximation by MLPs

Regardless of which method is used to optimise the weights, the basic function of an MLP is to learn and approximate the mapping a set of D input variables to a set of M output variables from a set of N examples A *learning example* in the training set can also be referred to as a *case* or as an *input pattern*, these terms are synonymous. The set of

examples used for learning is normally referred to as *the training set*. The number of input units an MLP has, is determined by the number of input variables (D) in the training set. Similarly, the number of output units an MLP has, is determined by the number of output variables (M) in the training set[1]. The number of hidden units an MLP has can be varied almost indefinitely.

The intrinsic mapping between the input and output variables, is also referred to as *the target function*. That is, it is the function, which the MLP, through training, is trying to learn to approximate. By systematically increasing the number of hidden units in an MLP, progressively more complex target functions can be potentially approximated. The upshot of this is that the degree to which an MLP is able to approximate a given target function can be systematically adjusted upwards by systematically increasing the number of hidden units.

MLPs have the important property of being universal function approximators (Reed & Marks 1999, Bishop 1995). That is MLPs are capable of approximating any bounded target function to any arbitrary degree of accuracy, if enough hidden units are available. All target functions, including arbitrarily determined functions, have a finite complexity. Since an MLP can have an infinite number of hidden units, and the complexity of the function mapping increases with the number of hidden units, then an MLP with "enough" hidden units can be used to map any input-output function exactly. Progressively increasing the number of hidden units ramps up the complexity of the MLP

[1] In most clinical decision making problems, and for all the problems considered this thesis M=1. That is there is only one output variable.

approximation mapping, until it matches the complexity level of the target function, in which case the approximation is at its best.

Bishop (1995) presents a proof of this proposition, which makes use of Kolmogorov's theorem, which inturn is a proof that every continuous function of several variables (for a closed and bounded input domain) can be represented as the superimposition of a small number of functions of one variable. An MLP with "enough" hidden units can map any required function exactly. With less than "enough" hidden units, an MLP approximates a given target function. As the number of hidden units increases, towards the number ("enough") which is required to map that given target function exactly, then the accuracy of that approximation increases.

The relationship of input patterns to output values, embodied in training set, is the target function, which an MLP learns to approximate during training. After successful training (by any method), the MLP is able to generate an appropriate output to a given input pattern. In the case where the mapping has been learned exactly then the exact output associated with a given input pattern can be generated on presentation of that input pattern. If the learned mapping is not exact then not all putative outputs will be the same as those "expected" in the training set. The better the approximation the better the match between the putative outputs of the MLP and their expected value from the training set.

For classification tasks, an MLP performs its input-output mapping by segmenting the input space into regions, using convex hulls or polygons as shown in figure A2.9. Given

enough hidden units an MLP can learn any arbitrary class membership mapping. That is mapping where the class of a case represented by a point in an input space is arbitrarily assigned, so that cases with the same class membership do not necessarily co-inhabit the same regions in the space. The function being approximated in this case is arbitrarily determined, but it can none-the-less be approximated (Reed & Marks, 1999, Bishop 1996, Ripley 1994). An example of such an arbitrary mapping would be to use data from a telephone book to train an MLP to output a telephone number in response to inputs consisting of a name and an address. In this example every case in the training set belongs to a class of its own, which is nominated by a telephone number.

However from the point of view of Clinical Decision Making in Psychiatry, the focus of thesis, we are not interested in arbitrary mappings (that come about as result of relationships to variables not contained in the dataset), but in the deterministic mappings that come about as the result of some causative process. That is the location of cases in the input space is determined by (or more correctly, is associated with) the class membership of that case. In such cases, where the mapping is non-arbitrary (or deterministic), it should be possible to generalise the learned function (mapping) to other cases drawn from the same population.

A2.15 Regularization

A classification problem presented to an MLP can be said to be "well posed" when:

- There is only one unique solution.
- The solution varies continuously (i.e. it is smooth) within the range of the given data

Well posed problems are relatively easy to solve and can yield a solution which generalizes well, conversely an ill posed problem is difficult to solve and obtained solutions will not generalize well.

The general idea behind regularization is to restate an ill posed problem as a well-posed problem (Reed & Marks, 1999). The strategy is to restrict the range of the search for solutions, only to those that are smooth (that is, functions where small changes in input values do not cause large variations in the output values). In practice, this is achieved by modifying the solution search algorithm (e.g BackProp), so that it penalizes solutions, which are not smooth. This introduces a bias in favour of choosing a smooth function as potential and final solutions.

Weight Decay

Large weights in an MLP cause sharp transitions in output values for small changes in inputs. Thus any potential solutions which contain large weights are no likely to generalize well, relative to other potential solutions that do not contain large weights. *Weight Decay* is a regularization technique used in MLPs. It penalizes large weights. So the MLP is biased towards solutions with smaller weights.

Weight Decay is implemented by adding a term (-βw) to the weight update equation (A2.11) so it becomes:

$$w^{new} = w^{old} - \eta\delta z - \beta w^{old} \qquad \text{(A2.22)}$$

Where β is a small fraction, which multiplied by the old value for the weight and the result subtracted from the old weight to reduce the size of the new weight by an amount proportionate to the size of the old weight.

The impact of weight decay is to reduce the size of weights. Large weights are reduced by large amounts, whilst small weights by only a small amount. This reduces the impact of outliers in the training set. Without weight decay the training algorithm, will recruit a weight(s) to accommodate an outlier. This leads to a value for a weight(s), which is larger than it would be if the outlier did not exist. Each time an outlier is processed by the training algorithm, to perform weight updates, the size of a particular weight(s) is increased until it is accommodated. Processing of the other cases uses other weights to accommodate them and leaves the weight(s) associated with the outlier unaffected.

Weight decay operates across all cases in the training set and on all weight updates, thus it will impact more on weights associated with outliers than on weights associated with non-outliers.

Jitter

Jitter is another regularization technique. Earlier in this chapter, it was reported that Reed & Marks (1999) and others have demonstrated that the addition of noise (random variation) in the values of the inputs reduces generalization for a given training set size. Thus it seems at first Paradoxical that jitter, a technique that involves the introduction of small amounts of noise (random variation) to input values in the training set, is a form of regularization, which can improve generalization.

The key difference is in how noise is added. In the procedure used by Reed & Marks (1999) to demonstrate the deleterious impact of noise, each input value was transformed (corrupted) by replacing the original value with another value, which is the original value +/- a random small amount, drawn from a normal distribution with a mean of 0 and a standard deviation, which is a small fraction of the standard deviation of the input variable. The size of the training set after the addition of noise is the same as before the addition of noise, but the input values have been corrupted by noise.

In Jitter, the original input cases are retained in the training set with their original input values unchanged. New derived cases are created by taking the input values of a case and replacing the original input values with another value, which is the original value +/- a

random small amount, drawn from a normal distribution with a mean of 0 and a standard deviation, which is a small fraction of the standard deviation of the input variable. The new derived cases are then added to the training set. Thus the new training set created by this procedure can be many times the size of the original training set and includes the original training set within it as a subset.

A jittered training set, implicitly embodies a "smoothness assumption" (Bishop 1995, Reed & Marks 1999). The impact of jitter is to smear the point value (in the input data space) of each case in the training set. The original value is the centroid of that smear. As a result more of the input space is accounted for by the presence of a training set point. This in turn requires less interpolation between training set points by the training algorithm. Because the smear around each original data point is normally distributed about that point the original deterministic function, which the training algorithm is trying to learn is preserved. The presence of the new smeared cases distributed and centered on the original case forces only smooth interpolations of the original training set cases and therefore biases against functions, which are not smooth.

Early Stopping

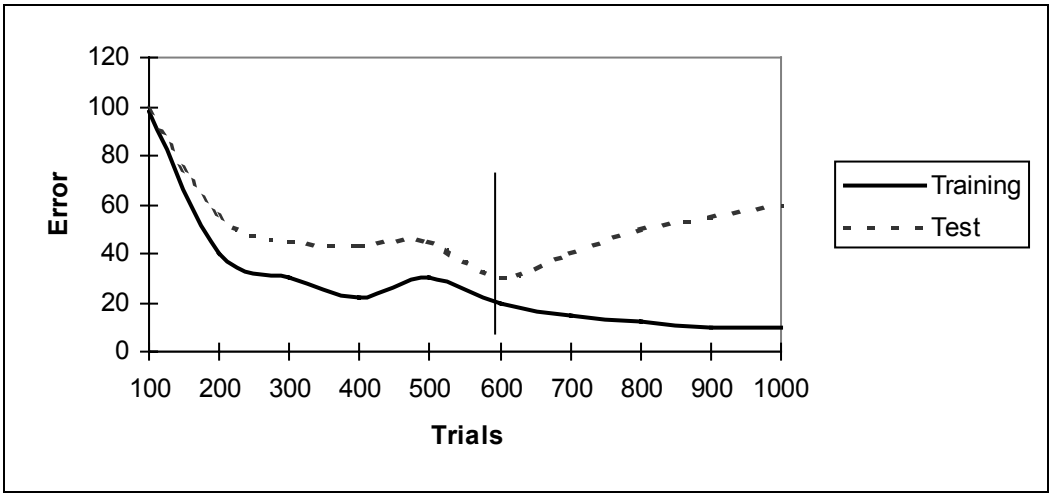


Figure A2.17 *Training Set Error and Test Set Error as a function of sequential trials of learning. Adapted from Reed & Marks 1999*

As learning proceeds, an MLP obtains progressively lower and lower levels of Training Set Error. This progress can be interrupted by short periods of rise in Training Set Error, attributable to local minima as seen in Figure A2.17 above, but in the long run the trend of Training Set learning curve is diminishing error, which becomes asymptotic to a value after a large number of trials. In Figure A2.17 above, training beyond 1000 trials is unlikely to yield any significant reductions in the Training Set Error.

Figure A2.17 also displays a curve for the Error obtained on a large independent Test Set as learning progresses. Initially the Test Set Error curve shadows the Training Set Error, but at a higher level of error. As learning progresses there eventually comes a point where the Test Set Error curve and the Training Set Error curve decouple. Beyond this point, the

Training Set Error continues to fall, whilst the Test Set Error begins to rise. This point is marked graph by a vertical line at just under 600 trials.

The significance of this behaviour is that the point of decoupling of the error curves is also the point that marks the beginning of *Overfitting* by an MLP. Overfitting is said to occur when the MLP has finished mapping the deterministic input-output function that is associated with the segregation of classes in the population and then goes on to map arbitrary input-output relationships caused by sampling variation, noise in the inputs and error in the class assignments.

To the left of the vertical line in Figure A2.17 the MLP is *Underfitting*. That is it has not yet completed mapping all aspects of the deterministic input-output relationship present in the population as a whole. To the right of the line the MLP is overfitting by mapping arbitrary relationships that exist only in the present Training Set sample. Maximum generalization by the MLP will occur if training is stopped at that pivot point and the weights from that point are used to classify new cases.

If a large independent Test Set exists, then it can be used to identify a point at which training should be stopped, by simultaneously measuring error on this Test Set. Once this point is identified the training can be re-run from the same starting point and then stopped at identified point. This procedure has been shown to produce good generalization (Bishop 1995, Reed & Marks 1999).

If a large independent Test Set is not available, a similar procedure is to "holdout" a fraction of the training set (say 25%) and use it as Test Set to identify the point where Overfitting begins, when training from the remaining cases. When the overfitting point is found, a note is made of the Training Set Error obtained at this point. The Holdout Set is then added back to the Training Set and the MLP is trained, not for a set number of trials, but until it obtains the same Training Set Error value that was noted at the identified overfitting point.

Both these procedures are called "Early Stopping" because they both stop training before the Training Set Error reaches a minima. The general idea is to maximize MLP's generalisation by preventing overfitting.

A2.16 Conditioning

The condition of the error surface of a Multi-layer Perceptron is a crucial determinant in finding a good solution, which can be generalized. The error surface is said to be ill-conditioned if it contains many features such as local minima, local maxima, saddle points and ravines which obscure the basin of attraction of the global minima. A well-conditioned error surface on the other hand would consist mainly of the basin of attraction of the global minima, with the error contours of that basin being circles, rather than ellipses or more complex curves.

BackProp and other gradient descent algorithms are local methods, which attempt to find the global minima by identifying and moving down the local error gradient. Therefore if the gradient direction at all points on the error surface is pointing at global minima, it will

be found easily from any starting point. If on the other hand, the presence of other features on the error surface distorts the basin of attraction of the global minimum, the local gradient at some points may be pointing in another direction away from the global minima. This will lead the algorithm to set off in a bad direction. In cases of mild ill-conditioning, this will slow down the algorithm because it will need to traverse terrain on the error surface until it reaches a point where the gradient points more directly at the global minima. In cases of severe ill-conditioning the algorithm may never find a good direction at any point and will fail altogether.

At any particular point, the condition of the local error surface can be estimated from the second derivatives of the weights with respect to error. The matrix of these derivatives for all the weights in an MLP neural network is called the Hesssian matrix (Bishop 1995, Sarle 1999, Reed & Marks 1999). First derivatives are an index of the gradient of a curve. Second derivatives are an index of the proximity of minima and maxima. If the local error surface is well conditioned, then the ratio of the largest and smallest eigenvalues of the Hessian matrix will be close to one. This indicates that there is strong basin of attraction around a single minima, which has circular error contours, because all the second derivatives are indicating the same minima. In such a situation all the first derivative gradients are pointing almost directly at the minima. Therefore a gradient descent algorithm like BackProp will accurately locate the global minima in a relatively small number of steps because it will essentially walk in a straight line towards it.

If the local error surface is ill-conditioned, that is the ratio of largest to smallest eigenvalues of the Hessian matrix is large, then this indicates that the error contours are locally convoluted by features such as local minima, maxima, saddle points and ravines. This inturn means that the gradients indicated by the first derivatives do not always point directly at the global minima. Therefore a gradient descent algorithm, like BackProp, will

tend to wander about this error surface and either take a long time (many steps) to find the basin of attraction of the global minima, or in severe cases of ill-conditioning wander aimlessly and never find it. Ill conditioning of MLP error surfaces, its evaluation by the Hessian matrix and remedies to the problem are studied by Sarle (1999). Discussion of the problem, the same and further suggestions are contained in Bishop (1995) and Reed & Marks (1999).

The solution to an ill-conditioned error surface is to prevent it or avoid it. There are several steps (Bishop 1995, Reed & Marks 1999, and Sarle 2000) that can be taken to prevent ill conditioning:

Firstly scale all the input variables to the same scale, using a technique like normalization (i.e. converting input values to z scores) or range rescaling. This has the effect of equalizing the influence of the scale of each input on the weights and makes error contours tend to be more circular rather than elliptical.

Secondly reduce, if possible, the number of input variables to a set, which contains only those that have strong relevance to the classification. The lower the dimensionality of the input space the better the condition of the error surface because the ratio of cases to weights is higher and thus weight estimates are better determined. Increasing the relevance of the inputs as a set improves the condition of the error surface because it increases the prominence of important features such as minima. Strategies for reducing the number of inputs and increasing the relevance of the input set are: a) eliminate variables that seem to bear no relationship to outputs or; b) use a data reduction technique such as Principal Components. Both methods have drawbacks, in terms of possibly discarding useful information. Research into more efficient techniques for selecting an

optimal set of inputs from candidates is an area of current interest in the literature (Reed & Marks, 1999).

A2.17 Cross-entropy Error Function

For classification problems, where the target values are 0 or 1, replace the MSE error function with the cross-entropy error function. The MSE error function implicitly assumes that the distribution of the target values is gaussian (i.e. normally distributed), which clearly it is not, because the distribution is binomial. The cross-entropy error function more closely approximates the true distribution of the target values and therefore gives a more accurate measure of error. Since the error surface is a plot of error against variations in the values of the weights, the error surface plotted from a cross-entropy error function is more accurate. The equation for the cross-entropy error function is:

$$E = -\sum_n \{t_n \ln z_n + (1 - t_n) \ln(1 - z_n)\} \qquad \text{(A2.24)}$$

where t = target value

z = output value

An interesting aspect of the cross-entropy error function is that when used in combination with the logistic activation function, the error signal which is produced for the back-error propagation algorithm is simply $z - t$.